Digital Electronics and VHDL

Practical 5 – Introducing Time

Table of Contents

[Introduction 3](#_Toc475429531)

[01 – Sequential VHDL – WHILE LOOPS 3](#_Toc475429532)

[TASK 01-01 3](#_Toc475429533)

[PRODUCTIVITY TIP! 4](#_Toc475429534)

[Self Assessment 5](#_Toc475429535)

[Comments 5](#_Toc475429536)

[Challenge 6](#_Toc475429537)

[Loop, exit and next 7](#_Toc475429538)

[TASK 01-02 Loop, EXIT and NEXT 7](#_Toc475429539)

[Comments 8](#_Toc475429540)

[The NEXT statement 9](#_Toc475429541)

[Challenge 9](#_Toc475429542)

[Task 01-03 Attributes 10](#_Toc475429543)

[02 - Synchronous Logic (first introduction) 11](#_Toc475429544)

[TASK 02-01 - wait statement 12](#_Toc475429545)

[TASK 02 - 02 - More Edge Detection and Multiple Architectures 13](#_Toc475429546)

[Appendix A – entities and architectures 15](#_Toc475429547)

[Entity 15](#_Toc475429548)

[Architecture 15](#_Toc475429549)

[APPENDIX B – PREDEFINED TYPES AND OPERATORS 16](#_Toc475429550)

[VHDL PREDEFINED TypeS 16](#_Toc475429551)

[VHDL Comparison operators 16](#_Toc475429552)

[VHDL INTEGER Operators 16](#_Toc475429553)

[VHDL Shifting Functions 16](#_Toc475429554)

[VHDL BINARY OPERATORS 17](#_Toc475429555)

[Appendix C - Concurrent statements 18](#_Toc475429556)

[When-Else 18](#_Toc475429557)

[SELECT 18](#_Toc475429558)

[Appendix D - TYPE and subtype DEFINTIONS 19](#_Toc475429559)

[Appendix E - Arrays 20](#_Toc475429560)

[APENDIX F - IEEE STD\_ULOGIC and STD\_LOGIC 21](#_Toc475429561)

[Appendix G - Structural statements 22](#_Toc475429562)

[component Declaration 22](#_Toc475429563)

[Instantiation 22](#_Toc475429564)

[Generate 22](#_Toc475429565)

[Generic Declarations 22](#_Toc475429566)

[Appendix H - Behavioural Statements 23](#_Toc475429567)

[process statement 23](#_Toc475429568)

[if statement 24](#_Toc475429569)

[If-ELSe 24](#_Toc475429570)

[if-elsif 24](#_Toc475429571)

[if-elsif-ELSE 24](#_Toc475429572)

[CASE 25](#_Toc475429573)

[LOOP 25](#_Toc475429574)

[FOR LOOP 25](#_Toc475429575)

[while loop 26](#_Toc475429576)

[NEXT and EXIT 26](#_Toc475429577)

[Appendix I - STANDARD ATTRIBUTES 27](#_Toc475429578)

[Type Attributes 27](#_Toc475429579)

[Array Attributes 27](#_Toc475429580)

[SIGNAL ATTRIBUTES 28](#_Toc475429581)

## **IMPORTANT UPDATES – READ THIS BEFORE YOU START**

With each year, there may be some minor updates to the tools. These are documented here. Please read this section before you begin.

### Target FPGA

The Cyclone II device sometimes featured in the videos is no longer supported in Quartus. We are currently using a **Cyclone® IV EP4CE22F17C6**

### Quartus II and the Vector Waveform Editor

Since version 13, Altera reinstated the vector waveform editor directly into Quartus II. This has not changed since v13. Some of the videos may make reference to an external tool. The tool is fundamentally the same, but now you can add a “vector waveform file” directly into your project from Quartus (File->New->… VWF)

To see how this has changed, please refer to the video “Using Quartus 13+ with VWF HD.mp4”

### Quartus User Interface

With each version that is released, the user interface can sometimes change in appearance. With version 16, the user interface has been noticeably re-skinned. However, the same basic functionality is still available from the menus as tool bar (the icons are now more colourful). To find the new icon, if necessary you can hover your mouse over the toolbar buttons, and you will see a text prompt. Given we only use a small subset of the buttons, it was not deemed necessary on this occasion to re-record the tutorial videos.

# Introduction

This practical continues to look at behavioural VHDL, and will address several new topics, including:

* **while loops**
* **loops**
* **branching within loops (next, exit, when)**
* **attributes**
* **introduction to synchronous logic**
* **introduction to configurations (multiple architectures)**

In this session, we complete our look at looping structures in VHDL, before taking a first loop at working with synchronous logic, that is, logic that is synchronised to a common clock.

# 01 – Sequential VHDL – WHILE LOOPS

Appendix H contains the syntax for a simple while-loop. As usual, we'll examine a while loop through a simple example. We begin with an entity "leading\_ones".

**entity** leading\_ones is

**port**

(

X : **in** std\_logic\_vector (15 downto 0); GO : **in** std\_logic;

Y : **out** std\_logic\_vector(3 downto 0) := "0000"

);

**end** **entity**;

### TASK 01-01

* Sketch a block diagram for this entity
* What type of inputs does this entity have (bit, vector, number of bits?)
* What type of outputs does this entity have (bit, vector, number of bits?)
* If you are unsure, show your answers to the tutor

This entity accepts a vectored input and will count the number of leading '1's. For example,

1100 0110 1000 1111 - has 2 leading ones

1111 0111 1111 1111 - has 4 leading ones

0111 0111 0101 1010 - has 0 leading ones

The architecture for this is given below:

**architecture** leading\_ones\_v1 **of** leading\_ones **is**

**begin**

**process** (X,GO) **is**

**variable** idx : integer range -1 to 15 := 15;

**variable** next\_bit : std\_logic;

**variable** res : integer;

**begin**

idx := 15;

res := 0;

**if** (GO = '1') **then**

next\_bit := X(idx);

|  |
| --- |
| **while** ((next\_bit = '1') **and** (idx>0)) **loop** res := res + 1; idx := idx - 1; next\_bit := X(idx); **end loop**; |

Y <= CONV\_STD\_LOGIC\_VECTOR(res, 4);

**else**

Y <= "ZZZZ";

**end if**;

**end** **process**;

**end** leading\_ones\_v1;

The high-lighted block shows the while loop. Note that this is all inside a **process block**. Statements inside a process block are known as **sequential statements**. VHDL is divided into concurrent and sequential statements.

### PRODUCTIVITY TIP!

Quartus offers the engineer some help with pre-defined templates. This is a useful feature that will save you a lot of time (and typing).

* Open a VHDL file in a Quartus project
* Right click and choose **Insert Template**
* Expand the VHDL node

You now have a few choices, including:

* Full designs are useful for getting the essential components of an entity and architecture
* Constructs is the other extreme. Here you will find basic VHDL language constructs, including concurrent and sequential VHDL.

Note that you don’t have to insert the selected template code – you can copy and paste sections as appropriate. I strongly advise you explore this feature of Quartus.

### Self Assessment and TASK 01-01

See if you can answer the following:

* What is the sensitivity list for this process block?
* Why is res a **variable** and not a **signal**?
* Why is res an integer and not of type std\_logic\_vector?
* What is the purpose of the function CONV\_STD\_LOGIC\_VECTOR and why do we need it?
* Can you explain, in words, how this architecture works?
* Now, open task 01-01, build and simulate
* Does it do what you expected?
* What is the purpose of the GO input?
* Do you think the GO input is edge or level triggered?
* **Advanced** - Is there a more *generic* way in the entity to initialise the following? (hint.. use *others)*Y : **out** std\_logic\_vector(3 downto 0) := "0000"

Note that with a while loop, the compiler must be able to see that the loop will finish in a finite time.

* Comment out the following lines

idx := 15;

res := 0;

* What happens when you try to build the code?
* **Advanced** - can you explain the error? (see over the page for an explanation) and/or discuss with the tutor

## Comments

Remember that sequential statements in a process block are a sequential way to describe parallel hardware. This is not C programming!

The while loop can only be synthesized if the compiler can see it only runs for a fixed period of time. In the help, there are some further guidelines:

[Quartus II VHDL Support](file:///C:\Altera\11.0sp1\quartus\common\help\webhelp\mergedProjects\hdl\vhdl\vhdl_list_support.htm) (see section 8)

One the loop statement it says:

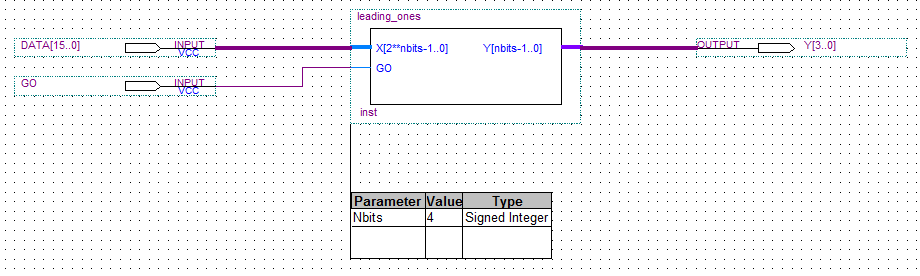
*" Supported. Loop statements must terminate within a fixed number of iterations during elaboration"*

Quartus II is a simulation **and** synthesis tool. When you build your code, it is trying to also synthesis your VHDL on a FPGA. Loops that never exit can be used in purely simulation environments (such as modelsim), but not in Quartus. Therefore, be aware that you will see documented in books a topic known as 'test benches' where non-synthesizable VHDL is used for testing. This code will probably not work in Quartus (you'll need to use another tool called modelsim which we cover later in the course).

### Challenge

The leading\_ones entity is set to work with a 16-bit input and 4-bit (to represent decimal 0..15) output.

* Using 'generics', modify the leading\_ones entity and architecture to allow the number of output / input bits to be changed



**entity** leading\_ones **is**

**generic**

(

Nbits : positive := 4

);

**port**

(

X : **in** std\_logic\_vector ((2 \*\* Nbits)-1 **downto** 0); GO : **in** std\_logic; Y : **out** std\_logic\_vector( (Nbits-1) **downto** 0)

);

end entity;

A solution is provided.

## Loop, exit and next

Is the appendix, you will also see a **loop** structure. For your convenience, it is repeated here:

**while** boolean-expression **loop**

*sequential-statement*

...

*sequential-statement*

**end loop**;

Note that this has no conditional statements for entering, repeating or exiting the loop. This is useful for simulation (only) where we might want to generate a clock signal for example. However, from the discussion above such statements are meaningless and therefore illegal for synthesis. Given that Quartus will attempt to synthesize your VHDL, it will not allow you to use such a loop without modification.

However, there are two statements you can use within any loop body that give you the ability to change how the loop behaves.

*[****next*** *[label] [****when*** *condition];*

*[****exit*** *[label] [****when*** *condition];*

We begin by looking at the **exit** statement

### TASK 01-02 Loop, EXIT and NEXT

* Open the project 01-02, build and simulate.
* Once again, this is the leading\_ones entity, but this time the architecture used a loop statement and an exit statement.

**entity** leading\_ones **is**

**port**

(

X : **in** std\_logic\_vector (15 downto 0);

GO : **in** std\_logic;

Y : **out** std\_logic\_vector(3 downto 0) := "0000"

);

**end entity**;

**architecture** leading\_ones\_v2 **of** leading\_ones **is**

**begin**

**process** (X,GO) **is**

**variable** idx : integer **range** -1 to 15 := 15;

**variable** next\_bit : std\_logic;

**variable** res : integer;

**begin**

idx := 15;

res := 0;

**if** (GO = '1') **then**

|  |
| --- |
| **loop**  next\_bit := X(idx);  **exit** **when** next\_bit = '0';  res := res + 1;  idx := idx - 1;  **exit** **when** idx = -1;  **end loop**; |

Y <= CONV\_STD\_LOGIC\_VECTOR(res, 4); **else**

Y <= "ZZZZ";

**end if**;

**end process**;

**end** leading\_ones\_v2;

Upon close examination, you can see the loop body.

* From this example, can you explain the purpose of the exit statement?
* Comment out the line exit when idx = -1; and try to re-build. What happens?

## Comments

The **exit** statement allows the engineer to break out of a loop. You could also use an **exit** and an **if**-statement together of the form

**if** <condition> **then**

**exit**;

**end if**;

## The NEXT statement

You can also use a **next** statement within a loop structure. The **next** statement **skips** the remaining logic inside a loop and jumps to the start of the loop body.

... **loop**

*<sequential-statement 1>*

*<sequential-statement 2>*

**next when** *<conditional statement>*

*<sequential-statement 3>*

**end loop;**

### Challenge

* Based on the code above, can you create a VHDL entity that simply counts the number of ones
* Use a loop and end-loop
* Use the next and exit statements as appropriate
* A solution is provided

Here is the entity

entity number\_of\_ones is

port

(

X : in std\_logic\_vector (15 downto 0);

GO : in std\_logic;

Y : out std\_logic\_vector(3 downto 0) := "0000"

);

end entity;

The output Y, when displayed in decimal, should indicate the number of ones in the input X. The output should be HighZ when GO is LOW.

## Task 01-03 Attributes

Much of VHDL , including types, signals, arrays and entities have what are called "attributes". These are properties essentially which you can read.

**Appendix I lists some of the attributes you can use in VHDL.**

There are two attributes we will consider in this session:

* The **range attribute** of an array
* The **event attribute** of a signal

Let's begin with the range attribute

* Open task 01-03
* Build and simulate
* Inspect the number\_of\_ones entity
* What type of variable is X?
* What is the range of X?
* Now inspect the architecture
* Note the following lines:

for i in X'range loop

if (X(i) = '1') then

res := res + 1;

end if;

end loop;

You can see the use of the attribute range. To access an attribute, you use the single-quote symbol ' (not to be confused with the single back-quote ` )

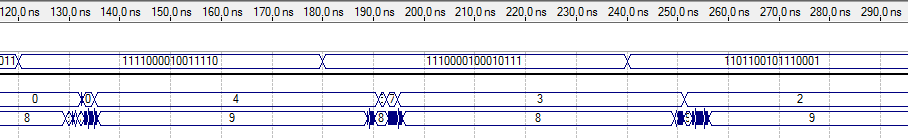
* Modify the leading\_ones architecture to use a **for-loop** and a **range** attribute.
* You might also want to use the **next** or **exit** statement to help you
* Build and simulate to demonstrate it is working

# 02 - Synchronous Logic (first introduction)

So far, we have only really considered combinational logic. This is said to be asynchronous as the inputs arrive at a random time, and the relative timing of the outputs depend only on propagation delay.

You will have probably noticed that the outputs of combinational logic also require time to settle to a steady state, and that there are temporary outputs which differ from the steady state (are logically incorrect).

Consider the previous example. A sample of the simulation output is shown below:



The yellow box regions illustrate periods where the outputs are (temporarily) erroneous, followed by a steady state results which is logically correct.

For these reasons, it is much harder to design reliable asynchronous logic of any complexity, especially where signal feedback is used (such as a state machine). Therefore, most sequential logic is designed to be synchronous:

**Synchronous circuits** a driven by a common clock. The output of logic devices are only sampled (read) on the **edge** of a clock. This can be the rising edge, falling edge or both.

From this point in the module, we will increasingly focus our attention on synchronous circuit design, implementation and testing. Of course, a significant part of synchronous systems is combinational logic.

To visualize what happens in synchronous logic, we rely heavily on timing diagrams (as shown above).

* We can use simulation to check the basic logic, check for initialized signals / bus contentions and even estimate propagation delays. We are limited by the speed and accuracy of a simulator of course.
* We can use an *on-chip logic analyzer* to test real logic – we will use **signal tap** to achieve this. Signal tap is a component that can be added to your design that can read internal signals, store them in RAM and output them for display in Quartus.

*In this module, you are expected to use BOTH simulation based testing and hardware testing (signal-tap)*

### TASK 02-01 - wait statement

* Open task 02-01, build and simulate
* Look at the VHDL source

**entity** sixteenbitcounter **is**

**port**

(

CLK : **in** std\_logic;

Y : **out** std\_logic\_vector(15 **downto** 0) := ('0', **others**=>'0')

);

**end entity**;

**architecture** counter16\_v1 **of** sixteenbitcounter **is**

**begin**

**process** **is**

**variable** x : natural := 0;

**begin**

**wait** **until** CLK = '1';

x := x + 1;

Y <= CONV\_STD\_LOGIC\_VECTOR(x, 16);

**end process**;

**end** counter16\_v1;

First of all, there is no sensitivity list specified in this case. We could have used one, but in this simple case, there was no motivation to do so. The line to focus on is the following:

**wait** **until** CLK = '1';

This line essentially says: suspend execution of this process block until the signal CLK **changes** to '1'. This "implies" a edge-detection. Personally, I don't like this style, but if you simulate it, you will see the behavior to be as expected.

You might think it would be clearer to also list CLK in a sensitivity list, this way, we are saying "only enter process block is CLK changes", therefore we are looking for a change (edge), not a level.

* Change the following line

process is

* to the following

process(CLK) is

* Try and build.

**If you wait on a signal, then essentially this makes it part of an implied sensitivity list.**

* Move the line with the wait statement to the bottom of the process block

begin

x := x + 1;

Y <= CONV\_STD\_LOGIC\_VECTOR(x, 16);

wait until CLK = '1';

end process;

* How does this change the result?

The location of the wait statement made no difference to the result. Although the statements are executed in sequence (as part of a hardware description), the semantics are unchanged. Furthermore, unlike variables, signals are not updated until a process block has completed[[1]](#footnote-1)

The **wait** statement is simply delaying the process block from exiting, and thus delaying any changes to the signal Y.

**Note** that Quartus will only allow a single wait instruction in a process block.

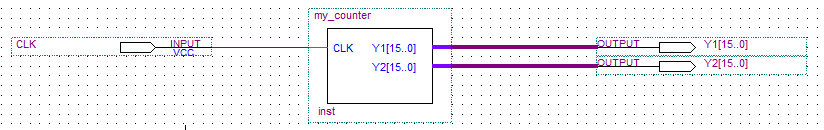
* **TASK** - change the counter to count on a falling edge
* Build and simulate

### TASK 02 - 02 - More Edge Detection and Multiple Architectures

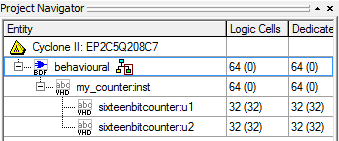
In this task, we look at another **attribute**, 'event'

* Open Task 02-02
* Build and simulate
* Inspect the file sixteenbitcounter.vhdl
* You will notice there is one entity and **three** architectures
  + each different architecture differs in the way the clock edge is detected.

Looking at the top level schematic, you will see a single entity with two counter outputs. One is driven of the rising edge of the clock, and the other from the falling edge.



* Can you find out which output is driven from which edge?
* The entity my\_counter is built using structural VHDL. It has two components, u1 and u2. This is also illustrated in the Project Navigator



* Look at the component declarations in the architecture, in particular these lines:

for all : CTR1 use entity work.sixteenbitcounter(counter16\_rising);

for all : CTR2 use entity work.sixteenbitcounter(counter16\_falling);

* + work refers to the current directory
  + sixteenbitcounter refers to an entity
  + the name in braces is the specific architecture you wish to use
* Examine the VHDL for these specific architectures

In one of these, counter16\_rising, we see the following code:

**process** (CLK) **is**

**variable** x : natural := 0;

**begin**

**if** (CLK'event **and** CLK = '1') **then**

x := x + 1;

Y <= CONV\_STD\_LOGIC\_VECTOR(x, 16);

**end if**;

**end process**;

where CLK is an input **signal**. From Appendix I, the **signal** **attribute** 'event' has the following meaning:

**S'EVENT** *is true if signal S has had an event this simulation cycle*

Note also that the signal CLK **is** listed in the sensitivity list, **but others could have been listed as well**. The if statement is checking to see if CLK was one of the signals (listed in the sensitivity list) that actually changed, and that it changed to a '1'

* Now contrast this to the architecture counter16\_rising\_v2 (not currently used)

**process** (CLK) **is**

**variable** x : natural := 0;

**begin**

**if** rising\_edge(CLK) **then**

x := x + 1;

Y <= CONV\_STD\_LOGIC\_VECTOR(x, 16);

**end if**;

**end process**;

This uses a convenient function rising\_edge[[2]](#footnote-2). Arguably this is more readable and a lot more memorable.

* Change the design so that U1 uses the architecture counter16\_rising\_v2
* Simulate and verify that U1 and U2 result in identical outputs

In the next session, we shall look at some case examples of common combinational and sequential building blocks - design recipes essentially.

# Appendix A – entities and architectures

## Entity

**entity** entity-name **is**

**port** (signal-names : mode signal-type [ := initial value ];

**port** (signal-names : mode signal-type [ := initial value ];

**port** (signal-names : mode signal-type [ := initial value ]

**end** entity-name;

**NOTE** – no semi-colon here!

|  |  |
| --- | --- |
| **Item** | **DESCRIPTION** |
| Entity-name | A name you choose, that matches the filename |
| Signal-names | A comma separated list of one or more input or output signals |
| Mode | This can be:  in – input  out – output  buffer – an output that can be read from within the architecture  inout – input or output, normally associated with tri-state outputs on PLD’s |
| Signal-type | The signal type. See Appendix B for pre-defined types. You can also create your own. |

## Architecture

**architecture** architecture-name if entity-name **is**

-- local variables, types etc…

type declarations

signal declarations

constant declarations

function definitions

procedure definitions

component declarations

**begin**

concurrent statement 1

concurrent statement 2

**end** architecture-name;

# APPENDIX B – PREDEFINED TYPES AND OPERATORS

## VHDL PREDEFINED TypeS

|  |  |
| --- | --- |
| **TYPE** | **DESCRIPTION** |
| bit | Single bit that takes values '0', '1' |
| bit\_vector | Vector (array) of bits |
| boolean | *true* or *false* |
| character | ISO 8-bit character |
| integer | Whole number between |
| real | Fractional numbers |
| severity\_level |  |
| string |  |
| time |  |

## VHDL Comparison operators

|  |  |
| --- | --- |
| **OPERATOR** | **DESCRIPTION** |
| = | Equals |
| /= | Not equals |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |

## VHDL INTEGER Operators

|  |  |
| --- | --- |
| **OPERATOR** | **DESCRIPTION** |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| Mod | Modulo division |
| Rem | Modulo remainder |
| Abs | Absolute value |
| \*\* | Exponentiation |

## VHDL Shifting Functions

|  |  |
| --- | --- |
| **OPERATOR** | **DESCRIPTION** |
| sll | Shift left logical |
| srl | Shift right logical |
| sla | Shift left arithmetic (preserve sign bit) |
| sra | Shift right arithmetic (preserve and copy sign bit) |
| rol | Rotate left |
| ror | Rotate right |

## VHDL BINARY OPERATORS

|  |  |
| --- | --- |
| **OPERATOR** | **DESCRIPTION** |
| and | AND |
| or | OR |
| nand | NAND |
| nor | NOR |
| xor | Exclusive OR |
| xnor | Exclusive NOR |
| not | Compliment (Inverter) |

# Appendix C - Concurrent statements

## When-Else

*signal-name* <= *expression* **when** *boolean-expression* **else**

*expression* **when** *boolean-expression* **else**

...

...

*expression* **when** *boolean-expression* **else**

*expression*;

## SELECT

**with** *expression* **select**

*signal-name* <= *signal-value* **when** *choices*,

*signal-value* **when** *choices*,

...

..

*signal-value* **when** *choices,*

*signal-value* **when****others**;

# Appendix D - TYPE and subtype DEFINTIONS

**type** *type-name* **is** (*value list*);

**subtype** *subtype-name* **is** *type-name* **range** *start* **to** *end*;

**subtype** *subtype-name* **is** *type-name* **range** *start* **downto** *end*;

**constant** *constant-name*: *type-name* := *value*;

# Appendix E - Arrays

**type** *type-name* **is** **array** (*start* **to** *end*) **of** *element-type*;

**type** *type-name* **is** **array** (*start* **downto** *end*) **of** *element-type*;

**type** type-name **is** **array** (*range-type*) **of** *element-type*;

**type** type-name **is** **array** (*range-type* **range** *start* **to** *end*) **of** *element-type*;

**type** type-name **is** **array** (*range-type* **range** *start* **downto** *end*) **of** *element-type*;

**type** *type\_name* **is array** (*type* **range <>) of** *element\_type*; -- unconstrained array

# APENDIX F - IEEE STD\_ULOGIC and STD\_LOGIC

**type** STD\_ULOGIC **is** ( 'U', -- uninitialized

'X', -- forcing unknown

'0', -- forcing 0

'1', -- forcing 1

'Z', -- High Impedance

'W', -- Weak unknown

'L', -- Weak 0

'H', -- Weak 1

'-', -- Don't care

);

**subtype** STD\_LOGIC **is resolved** STD\_ULOGIC;

-- and the vectors

**type** STD\_ULOGIC\_VECTOR **is array** (natural **range** <>) **of** STD\_ULOGIC;

**type** STD\_LOGIC\_VECTOR **is array** (natural **range** <>) **of** STD\_LOGIC;

# Appendix G - Structural statements

## component Declaration

**component** *component-name*

**port** ( *signal-names* : *mode* *signal-type*;

*signal-names* : *mode* *signal-type*;

...

*signal-names* : *mode* *signal-type )*;

**end component**;

## Instantiation

*label: component-name* **port map** (*signal1, signal2, ..., signaln*);

*label: component-name* **port map** (*port1=>signal1, port2=>signal2, ..., portn=>signaln*);

## Generate

*label*: **for** *identifier* **in** *range* **generate**

*concurrent-statement*

**end generate;**

## Generic Declarations

**generic** ( *constant-names* : *constant-type*;

*constant-names* : *constant-type*;

...

*constant-names* : *constant-type*);

# Appendix H - Behavioural Statements

## process statement

**process** (*signal-name, signal-name, ..., signal-name*)

*type declarations*

*variable declarations*

*constant declarations*

*function definitions*

*procedure definitions*

**begin**

*sequential statement*

*sequential statement*

*...*

*sequential statement*

**end process;**

## if statement

**if** *boolean-expression* **then** *sequential-statements*

end if;

## If-ELSe

**if** *boolean-expression* **then** *sequential-statements*

**else** *sequential-statements*

**end if;**

## if-elsif

**if** *boolean-expression* **then** *sequential-statements*

**elsif** *boolean-expression* **then** *sequential-statements*

**elsif** *boolean-expression* **then** *sequential-statements*

...

**elsif** *boolean-expression* **then** *sequential-statements*

**end if;**

## if-elsif-ELSE

**if** *boolean-expression* **then** *sequential-statements*

**elsif** *boolean-expression* **then** *sequential-statements*

**elsif** *boolean-expression* **then** *sequential-statements*

...

**elsif** *boolean-expression* **then** *sequential-statements*

**else** *sequential-statements*

**end if;**

## CASE

**case** expression **is**

**when** choices => sequential-statements

**when** choices => sequential-statements

...

**when** choices => sequential-statements

**end case;**

## LOOP

**loop**

*sequential-statement*

*sequential-statement*

*...*

*sequential-statement*

*[****next*** *[label] [****when*** *condition];*

*[****exit*** *[label] [****when*** *condition];*

**end loop**;

## FOR LOOP

**for** identifier **in** range **loop**

*sequential-statement*

*sequential-statement*

...

*sequential-statement*

*[****next*** *[label] [****when*** *condition];*

*[****exit*** *[label] [****when*** *condition];*

**end loop**;

## while loop

**while** boolean-expression **loop**

*sequential-statement*

*sequential-statement*

...

*sequential-statement*

*[****next*** *[label] [****when*** *condition];*

*[****exit*** *[label] [****when*** *condition];*

**end loop**;

## NEXT and EXIT

The **next** statement skips to the next iteration of the loop.

The **exit** statement skips the reset of the statements in the loop body and breaks out of the loop.

# Appendix I - STANDARD ATTRIBUTES

*(from http://www.cs.umbc.edu/portal/help/VHDL/attribute.html, accessed 4/11/2010)*

**T** represents any type

**A** represents any array or constrained array type

**S** represents any signal

**E** represents a named entity

## Type Attributes

**T'BASE** *is the base type of the type T*

**T'LEFT** *is the leftmost value of type T (Largest if using downto)*

**T'RIGHT** *is the rightmost value of type T. (Smallest if using downto)*

**T'HIGH** *is the highest value of type T*

**T'LOW** *is the lowest value of type T*

**T'ASCENDING** *is Boolean true if range and T defined with 'to'*.

**T'IMAGE(X)**  *is a string representation of X that is of type T*

**T'VALUE(X)**  *is a value of type T converted from the string X*

**T'POS(X)** *is the integer position of X in the discrete type T*

**T'VAL(X)**  *is the value of discrete type T at integer position X*

**T'SUCC(X)** *is the value of discrete type T that is the successor of X*

**T'PRED(X)** *is the value of discrete type T that is the predecessor of X*

**T'LEFTOF(X)** *is the value of discrete type T that is left of X*

**T'RIGHTOF(X)** *is the value of discrete type T that is right of X*

## Array Attributes

**A'LEFT** *is the leftmost subscript of array A or constrained array type*

**A'LEFT(N)** *is the leftmost subscript of dimension N of array A*

**A'RIGHT** *is the rightmost subscript of array A or constrained array type*

**A'RIGHT(N)** *is the rightmost subscript of dimension N of array A*

**A'HIGH** *is the highest subscript of array A or constrained array type*

**A'HIGH(N)** *is the highest subscript of dimension N of array A*

**A'LOW**  *is the lowest subscript of array A or constrained array type*

**A'LOW(N)** *is the lowest subscript of dimension N of array A*

**A'RANGE** *is the range A'LEFT to A'RIGHT or A'LEFT downto A'RIGHT*

**A'RANGE(N)** *is the range of dimension N of A*

**A'REVERSE\_RANGE** *is the range of A with to and downto reversed*

**A'REVERSE\_RANGE(N)** *is the REVERSE\_RANGE of dimension N of array A*

**A'LENGTH** *is the integer value of the number of elements in array A*

**A'LENGTH(N)** *is the number of elements of dimension N of array A*

**A'ASCENDING** *is boolean true if range of A defined with to*

**A'ASCENDING(N)** *is boolean true if dimension N of array A defined with to*

## SIGNAL ATTRIBUTES

**S'DELAYED(t)** *is the signal value of S at time now - t*

**S'STABLE** *is true if no event is occurring on signal S*

**S'STABLE(t)** *is true if no event has occurred on signal S for t units of time*

**S'QUIET** *is true if signal S is quiet (no event this simulation cycle)*

**S'QUIET(t)** *is true if signal S has been quiet for t units of time*

**S'TRANSACTION** *is a bit signal, the inverse of previous value each cycle S is active*

**S'EVENT** *is true if signal S has had an event this simulation cycle*

**S'ACTIVE** *is true if signal S is active during current simulation cycle*

**S'LAST\_EVENT** is the time since the last event on signal S

**S'LAST\_ACTIVE** *is the time since signal S was last active*

**S'LAST\_VALUE** *is the previous value of signal S*

**S'DRIVING** *is false only if the current driver of S is a null transaction*

**S'DRIVING\_VALUE** *is the current driving value of signal S*

**E'SIMPLE\_NAME** *is a string containing the name of entity E*

**E'INSTANCE\_NAME** *is a string containing the design hierarchy including E*

**E'PATH\_NAME** *is a string containing the design hierarchy of E to design root*

1. remember - a process block itself is a concurrent statement which simulates in zero time. [↑](#footnote-ref-1)
2. There is also a falling\_edge function of course! [↑](#footnote-ref-2)